Generating Training Data for Learning Linear Composite Dispatching Rules for Scheduling

Helga Ingimundardottir and Thomas Philip Runarsson

School of Engineering and Natural Sciences, University of Iceland
hei2@hi.is and tpr@hi.is

Abstract. A supervised learning approach to generating composite linear priority dispatching rules for scheduling is studied. In particular we investigate a number of strategies for generating training data for learning a linear dispatching rule using preference learning. The results show that generating training data set from optimal solutions only is not as effective as when suboptimal solutions are added to the set. Furthermore, different strategies for creating preference pairs is investigated as well as sub-optimal solution trajectories. The different strategies are investigates on some 2000 randomly generated problem instances using two different problems generator settings.

1 Introduction

The job-shop scheduling problem (JSP) deals with the allocation of tasks of competing resources where the goal is to optimise a single or multiple objectives, in particular minimising a schedule's maximum completion time, i.e. the makespan. Due to difficulty in solving this problem heuristics are common applied. Perhaps the simplest approach to generating good solutions to the JSP is by applying dispatching rules [1]. For example, dispatching a job which has the most work remaining (MWR). Composites of such simple rules can perform significantly better [2]. As a consequence a linear composite of dispatching rule was presented by the authors in [3]. There the goal was to learn a set of weights, $\mathbf{w}$ via logistic regression such that

$$h(\mathbf{x}_j) = \big\langle \mathbf{w} \cdot \phi(\mathbf{x}_j) \big\rangle, \tag{1}$$

yields the preference estimate for dispatching the job j that corresponds to post-decision state $\mathbf{x}_j$, where $\phi(\mathbf{x}_j)$ denotes the feature mapping. The features may correspond to a dispatching rule, for example the single feature $\phi_1(\mathbf{x}_j)$ would correspond be the work remaining heuristic if $h(\mathbf{x}_j) > h(\mathbf{x}_i)$, $\forall i$ are jobs with less work remaining than job j.

The weights in [3] then found using supervised learning, where the training data was created from optimal solutions of randomly generated problem instances. As an alternative would be to minimizing the mean makespan directly using a brute force search such as the CMA-ES [4]. This actually results in a better linear composite priority dispatching rules. The nature of the CMA-ES search is to explore suboptimal routes until it converges to an optimal one.

Implying that the previous approach of only looking into one optimal route may not produce a sufficient rich training set. That is, the training set should incorporate a more complete knowledge on *all* possible preferences, i.e. make also the distinction between suboptimal and sub-suboptimal features, etc. This would require a Pareto ranking of preferences which can be used to make the distinction to which feature sets are equivalent, better or worse – and to what degree, i.e. by giving a weight to the preference. This would results in a very large training set, which of course could be re-sampled in order to make it computationally feasible to learn. Here we will investigate a number of different ranking strategies for creating preference pairs.

Alternatively, training data could be generated using sub-optimal solution trajectories. For instance [5] used decision trees to 'rediscover' the LPT single priority based dispatching rule by using the dispatching rule to create its training data. The limitations of using heuristics to label the training data is that the learning algorithm will mimic the original heuristic (both when it works poorly and well on the problem instances) and does not consider the real optimum. In order to learn heuristics that can outperform existing heuristics, then the training data needs to be correctly labelled. This drawback is confronted in [6,7,8] by using an optimal scheduler, computed off-line. Here we will both follow optimal and sub-optimal solution trajectories, but for each partial solution the preference pair will be labelled correctly by solving the partial solution to optimality using a commercial software package [9]. For this study only MWR, the promising single priority dispatching rule (see [10]) for the given data distributions, and the CMA-ES found linear dispatching rule will be deemed worthwhile for generating suboptimal trajectories.

In summary, the paper considers two main aspects of the generation of the training data,

1. how preference pair are created at each decision stage, and
2. which solution trajectorie(s) should be sampled. That is, optimal, random, sub-optimal based on a good heuristic, etc.

The paper first illustrated how the JSP problem can be seen as a decision tree where the depth of the tree corresponds to the number of job dispatches needed to form a complete schedule. The feature space is also introduced and how optimal dispatches and sub-optimal dispatches are labelled as each node in the tree. This is followed by a section detailing the strategies investigated in this paper for selecting preference pairs ranking and sampling solution trajectories. We then perform an extensive study comparing these strategies, followed by a conclusion and summary of main results.

2 JSP tree representation

When building a complete JSP schedule $\ell = n \cdot m$ (n jobs and m machines) dispatches must be made sequentially. A job is placed at the earliest available time slot for its next machine, whilst still fulfilling constraints that each machine

Table 1. Feature space $\mathcal{F}$ for JSP where job J_j on machine M_a given the resulting temporal schedule after dispatching (j, a).

ϕ	Feature description
ϕ_1	processing time
ϕ_2	start-time
ϕ_3	end-time
ϕ_4	when machine is next free
ϕ_5	current makespan
ϕ_6	work remaining
ϕ_7	most work remaining
ϕ_8	slack time for machine
ϕ_9	slack time for all machines
ϕ_{10}	slack weighted w.r.t. number of tasks assigned
ϕ_{11}	time job had to wait
ϕ_{12}	size of slot created by assignment
ϕ_{13}	total processing time for job

can handle at most one job at each time, and jobs need to have finished their previous machines according to its machine order. Unfinished jobs are dispatched one at a time according to some heuristic. After each dispatch[1] the schedule's current features (cf. Table 1) are updated based on the half-finished schedule. Fig. 1 shows how the first two dispatches could be executed for a six-job six-machine job-shop scheduling problem, with the machines, $a \in \{M_1, ..., M_6\}$, on the vertical axis and the horizontal axis yields the current makespan. The next possible dispatches are denoted as dashed boxes with the job index j within and its length corresponding to p_{ja}. In the top layer one can see an empty schedule. In the middle layer one of the possible dispatches from the layer above is fixed, and one can see the resulting schedule, i.e. what are the next possible dispatches given this scenario? This sort of tree representation is similar to *game trees* [11] where the root node denotes the initial, i.e. empty, schedule and the leaf nodes denote the complete schedule, therefore the distance k from an internal node to the root yields the number of operations already dispatched. Traversing from root to leaf node one can obtain a sequence of dispatches that yielded the resulting schedule, i.e. the sequence indicates in which order the tasks should be dispatched for that particular schedule.

However, one can easily see that this sequence of task assignments is by no means unique. Inspecting a partial schedule further along in the dispatching process such as in Fig. 1 (top layer), then let's say J_1 would be dispatched next, and in the next iteration J_2. Now this sequence would yield the same schedule as if J_2 would have been dispatched first and then J_1 in the next iteration, i.e. these are non-conflicting jobs. Which indicates that some of the nodes in the tree can merge. In the meantime the state of the schedules are different and thus their features, although they manage to merge with the same (partial) schedule at a

[1] Dispatch and time step are used interchangeably.

later date. In this particular instance one can not infer that choosing J_1 is better and J_2 is worse (or vice versa) since they can both yield the same solution.

Furthermore, in some cases there can be multiple optimal solutions to the same problem instance. Hence not only is the sequence representation 'flawed' in the sense that slight permutations on the sequence are in fact equivalent w.r.t. the end-result, but varying permutations on the dispatching sequence (given the same partial initial sequence) can result in very different complete schedules but the same makespan, and thus same deviation from optimality ρ defined by (2), which is the measure under consideration. Care must be taken in this case that neither resulting features are labelled as undesirable or suboptimal. Only the

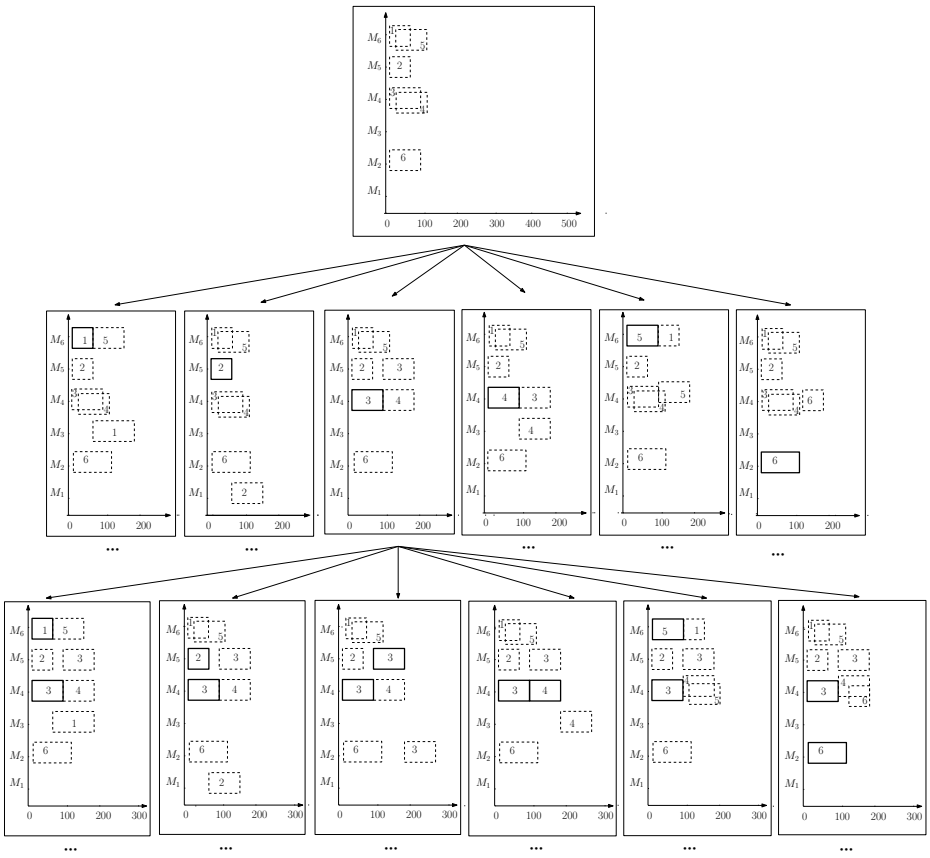

Fig. 1. Partial Tree for job-shop scheduling problem for the first two dispatches. Top layer depicts all possible dispatches (dashed) for an empty schedule. Middle layer depicts all possible dispatches given that one of the dispatches from the layer above has been executed (solid). Bottom layer depicts when job J_3 on machine M_4 has been chosen to be dispatched from the previous layer, moreover it depicts all possible next dispatches from that scenario.

resulting features from a dispatch resulting in a suboptimal solution should be labelled undesirable.

The creation of the tree for job-shop scheduling can be done recursively for all possible permutation of dispatches, in the manner described above, resulting in a full n-ary tree of height $\ell = n \cdot m$. Such an exhaustive search would yield at the most n^ℓ leaf nodes (worst case scenario being that no sub-trees merge). Now, since the internal vertices, i.e. partial schedules, are only of interest to learn,[2] the number of those can be at the most ${}^{n^\ell - 1}/_{n-1}$. Even for small dimensions of n and m the number of internal vertices are quite substantial and thus computationally expensive to investigate them all.

The optimum makespan is known for each problem instance. At each time step (i.e. layer of the tree) a number of feature pair are created, they consist of the features ϕ_o resulting from optimal dispatches $o \in \mathcal{O}^{(k)}$, versus features ϕ_s resulting from suboptimal dispatches $s \in \mathcal{S}^{(k)}$ at time k. Note, $\mathcal{O}^{(k)} \cup \mathcal{S}^{(k)} = \mathcal{R}^{(k)}$ and $\mathcal{O}^{(k)} \cap \mathcal{S}^{(k)} = \emptyset$. In particular, each job is compared against another job of the ready-list, $\mathcal{R}^{(k)}$, and if the makespan differs, i.e. $C_{\max}^{(s)} \gneq C_{\max}^{(o)}$, an optimal/suboptimal pair is created, however if the makespan would be unaltered the pair is omitted since they give the same optimal makespan. This way, only features from a dispatch resulting in a suboptimal solution is labelled undesirable.

The approach taken here is to verify analytically, at each time step, by fixing the current temporal schedule as an initial state, whether it can indeed *somehow* yield an optimal schedule by manipulating the remainder of the sequence. This also takes care of the scenario that having dispatched a job resulting in a different temporal makespan would have resulted in the same final makespan if another optimal dispatching sequence would have been chosen. That is to say the data generation takes into consideration when there are multiple optimal solutions to the same problem instance.

3 Selecting preference pairs

At each dispatch iteration k a number of preference pairs are created which can then be multiplied by the number of problem instance N created. A separate data set is deliberately created for each dispatch iterations, as the initial feeling is that dispatch rules used in the beginning of the schedule building process may not necessarily be the same as in the middle or end of the schedule. As a result there are ℓ linear scheduling rules for solving a $n \times m$ job-shop. Defining the size of the preference set as $l = |S|$ (cf. [3]). If l is too large, then re-sampling may need to be done in order for the ordinal regression to be computationally feasible.

[2] The root is the empty initial schedule and for the last dispatch there is only one option left to dispatch, so there is no preferred 'choice' to learn.

3.1 Ranking strategies

The following ranking strategies were implemented for adding preference pairs to S,

S_b all optimum rankings r_1 versus all possible sub-optimum rankings r_i, $i \in \{2, \ldots, n'\}$, preference pairs are added, i.e. same basic set-up as in [3].
S_f full subsequent rankings, i.e. all possible combinations of r_i and r_{i+1} for $i \in \{1, \ldots, n'\}$, preference pairs are added.
S_p partial subsequent rankings, i.e. sufficient set of combinations of r_i and r_{i+1} for $i \in \{1, \ldots, n'\}$, are added to the training set – e.g. in the cases that there are more than one operation with the same ranking, only one of that rank is needed to compared to the subsequent rank. Note that $S_p \subset S_f$.

where $r_1 > r_2 > \ldots > r_{n'}$ $(n' \leq n)$ are the rankings of the ready-list, $\mathcal{R}^{(k)}$, at time step k.

3.2 Trajectory sampling strategies

The following trajectory sampling strategies were explored for adding preference pairs to S,

S^{opt} at each dispatch some (random) optimal task is dispatched.
S^{cma} at each dispatch the task corresponding to highest priority, computed with fixed weights $\mathbf{w}$, which were obtained by optimising the mean for (2) with CMA-ES.
S^{mwr} at each dispatch the task corresponding to most work remaining is dispatched, i.e. following the simple dispatching rule MWR.
S^{rnd} at each dispatch some random task is dispatched.

In the case of S^{mwr} and S^{cma} it is sufficient to explore each trajectory exactly once for each problem instance. Whereas, for S^{opt} and S^{rnd} there can be several trajectories worth exploring, however, only one is chosen (at random). It is noted that since the number of problem instances N is large, it is deemed sufficient to explore one trajectory for each instance, in those cases as well.

4 Experimental study

For N problem instances generated using n jobs and m machines for processing times following the same distribution and random σ permutations of job orderings. The optimum makespan is denoted $C^{\text{opt}}_{\max}$, and the makespan obtained from the linear learning model by $C^{\text{model}}_{\max}$. Since the optimal makespan varies between problem instances the performance measure is the following,

$$\rho = \frac{C^{\text{model}}_{\max} - C^{opt}_{\max}}{C^{\text{opt}}_{\max}} \cdot 100\% \tag{2}$$

which indicates the percentage relative deviation from optimality.

To test the validity of different ranking and strategies from section 3, a training set of $N_{\text{train}} = 500$ and $N_{\text{test}} = 500$ problem instances of 6×5 job-shop for several problem space distributions, namely $\mathcal{P}_1$ and $\mathcal{P}_2$, where processing are drawn from a uniform distribution $\mathcal{U}(10, 100)$ and $\mathcal{U}(50, 100)$ respectively. The size of the preference set, S, for different trajectory and ranking strategies is depicted in Fig. 2 and 3, for $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively.

A linear ordinal regression model was created for each preference set, S, for problem space $\mathcal{P}_1$. A box-plot with the results of percentage relative deviation from optimality, ρ, defined by (2), is presented in Fig. 4 and 5. Fig. 4 depicts different ranking strategies for a fixed trajectory, whereas Fig. 5 depicts different trajectory strategies for a fixed ranking. From the figures it is apparent there can be a performance edge gained by implementing a particular ranking or trajectory strategy, moreover the behaviour is analogous across different disciplines. Similarly, Fig. 6 and 7 for problem space $\mathcal{P}_2$.

Note that S_{all} denotes that all rankings were explored, i.e. $S_{all} = S_b \cup S_f \cup S_p$. Similarly, $S^{all} = S^{opt} \cup S^{cma} \cup S^{mwr} \cup S^{rnd}$ for all trajectories.

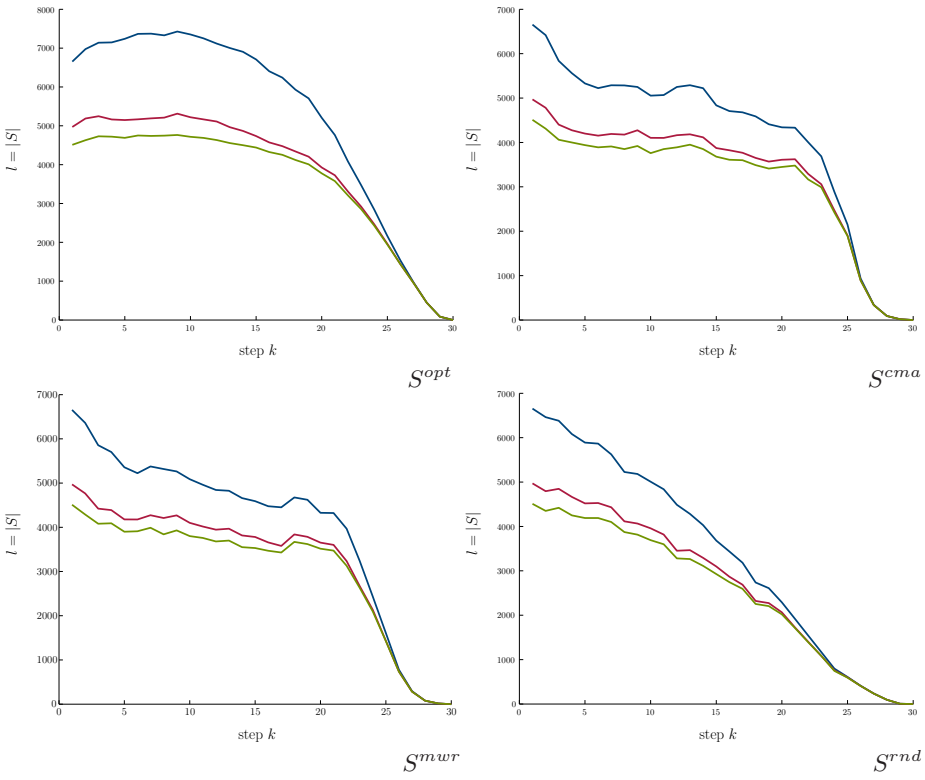

Fig. 2. Size of preference set, l for different trajectory and ranking strategies (S_b in blue, S_f in red, S_p in green), given problem space $\mathcal{P}_1$, where $N_{\text{train}} = 500$.

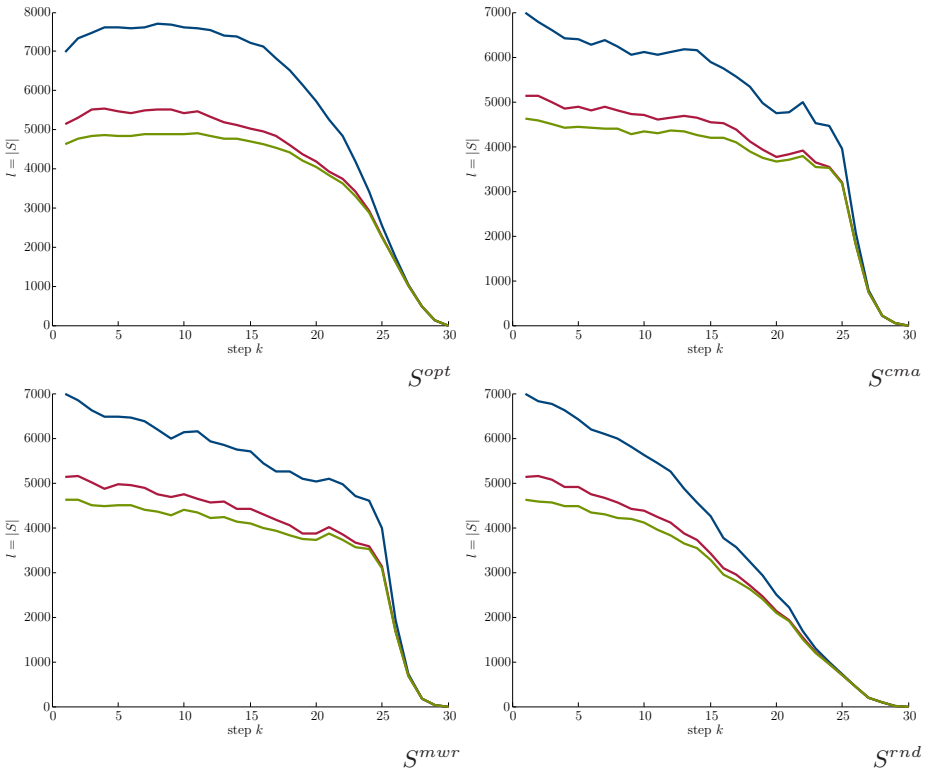

Fig. 3. Size of preference set, l for different trajectory and ranking strategies (S_b in blue, S_f in red, S_p in green), given problem space $\mathcal{P}_2$, where $N_{\text{train}} = 500$.

4.1 Ranking strategies

There is no statistical difference between S_f and S_p ranking-schemes across all disciplines (cf. Fig. 4 and 6), which is expected since S_f is designed to contain the same preference information as S_f. However neither of the Pareto ranking-schemes outperform the original S_b set-up from [3]. The results hold for both test and training sets.

Combining the ranking schemes, S_{all}, improves the individual ranking-schemes across all disciplines, except in the case of $S_b^{opt}\big|_{\mathcal{P}_1}$ and $S_b^{rnd}\big|_{\mathcal{P}_2}$, in which case there were no statistical difference. Now, for the test set, the results hold, however there is no statistical difference between S_b and S_{all} for most trajectories $\{S^{opt}, S^{cma}, S^{rnd}\}\big|_{\mathcal{P}_1}$ and $\{S^{opt}, S^{rnd}\}\big|_{\mathcal{P}_2}$. Now, whereas a smaller preference set is preferred, its opted to use the S^b ranking scheme.

Moreover, it is noted that the learning algorithm is able to significantly outperform the original heuristics, MWR and CMA-ES (white), used to create the training data S^{mwr} (grey) and S^{cma} (yellow), respectively (cf. Fig. 4 and 6).

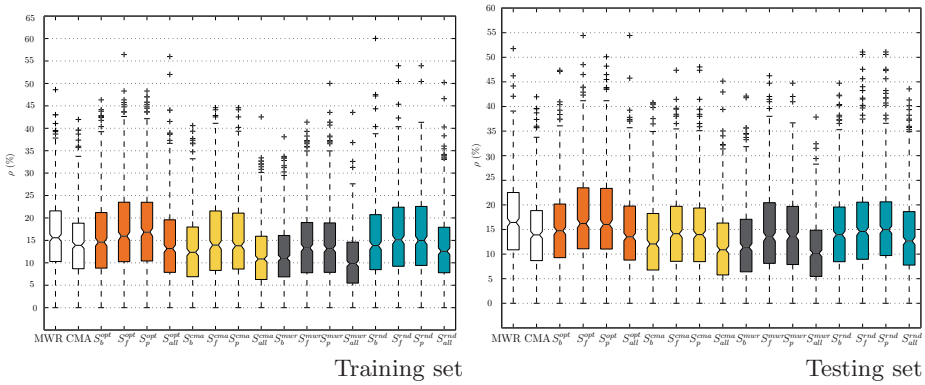

Fig. 4. Box-plot of results for linear ordinal regression model trained on various preference sets using problem space $\mathcal{P}_1$. Note that same trajectory schemes are colour-coded the same.

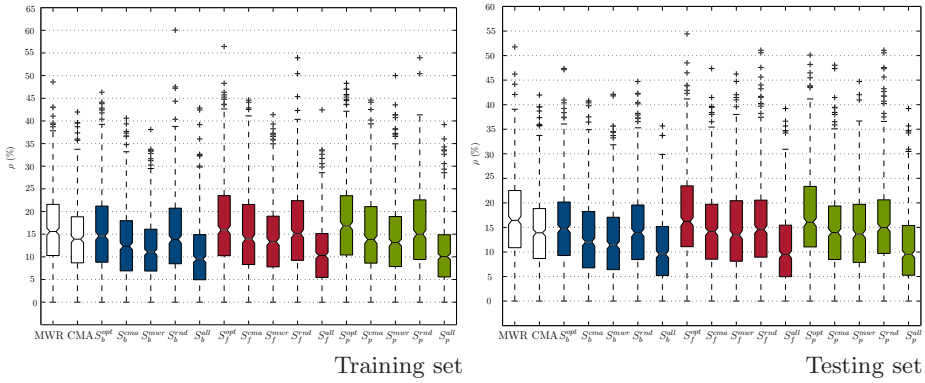

Fig. 5. Box-plot of results for linear ordinal regression model trained on various preference sets using problem space $\mathcal{P}_1$. Note that same ranking schemes are colour-coded the same.

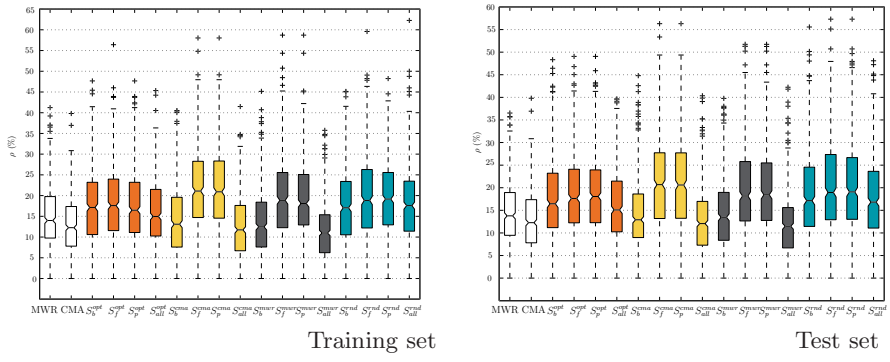

Fig. 6. Box-plot of results for linear ordinal regression model trained on various preference sets using problem space $\mathcal{P}_2$.

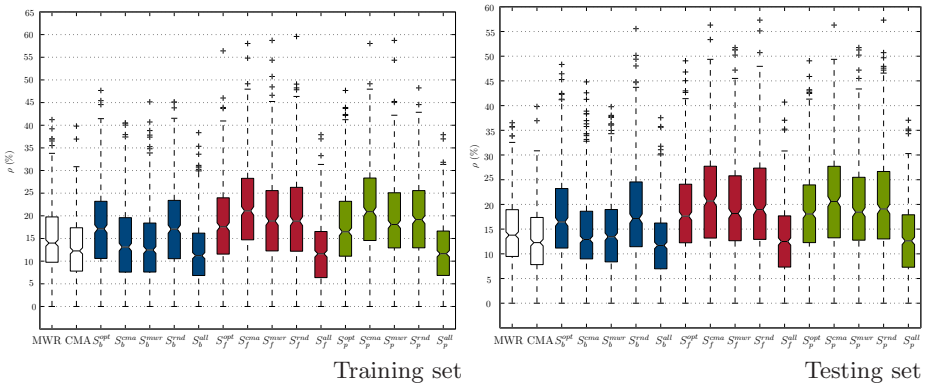

Fig. 7. Box-plot of results for linear ordinal regression model trained on various preference sets using problem space $\mathcal{P}_2$.

For both $\mathcal{P}_1$ and $\mathcal{P}_2$, linear ordinal regression models based on S^{mwr} are significantly better than MWR, irrespective of the ranking schemes. Whereas the fixed weights found via CMA-ES are only outperformed by linear ordinal regression models based on $\{S_b^{cma}, S_{all}^{cma}\}$. This implies that ranking scheme needs to be selected appropriately. Result hold for the test data.

4.2 Trajectory sampling strategies

Learning preference pairs from a good scheduling policies, such as S^{cma} and S^{mwr}, gave considerably more favourable results than tracking optimal paths (cf. Fig. 4 and 4). Suboptimal routes are preferred when dealing with $\mathcal{P}_1$ (for all ranking schemes), however when encountering $\mathcal{P}_2$ the choice of ranking schemes can yield the exact opposite.

It is particularly interesting there is no statistical difference between S^{opt} and S^{rnd} for both $\{S_b, S_f\}\big|_{\mathcal{P}_1}$ and $\{S_b, S_f, S_p\}\big|_{\mathcal{P}_2}$ ranking-schemes. That is to say, tracking optimal dispatches gives the same performance as completely random dispatches. This indicates that exploring only optimal trajectories can result in a training set where the learning algorithm is inept to determine good dispatches in the circumstances when newly encountered features have diverged from the learned feature set labelled to optimum solutions.

Finally, S^{all} gave the best combination across all disciplines. Adding suboptimal trajectories with the optimal trajectories gives the learning algorithm a greater variety of preference pairs for getting out of local minima.

4.3 Following CMA-ES guided trajectory

The rational for using the S^{cma} strategy was mostly due to the fact a linear classifier is creating the training data (using the weights found via CMA-ES optimisation), hence the training data created should be linearly separable,

which in turn should boost the training accuracy for a linear classification learning model. However, this strategy is easily outperformed by the single priority based dispatching rule MWR guiding the training data collection, S^{mwr}.

Let's inspect the CMA-ES guided training data more closely, in particular the linear weights for (1). The weights are depicted in Fig. 8 and 9 for problem space $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. The original weights found via CMA-ES optimisation, that are used to guide the collection of training data, are depicted in red and weights obtained by the linear classification model for S_b^{cma} are depicted in blue.

From the CMA experiments it is clear that a lot of weight is applied to the w_6 that corresponds implementing MWR, yet the existing weights for other features diverges the training data from a more "better" training set to learn. Arguably, it might be due to the computational cost involved in implementing CMA-ES, meaning that even after 1,500 function evaluations the method might still be far from optimum weights. It might also be an artefact due the fact the training set during the CMA-ES search is different to the data generation described in Sec. 3 is completely different, hence the different scaling parameters for the features might influence the results. Moreover, the CMA-ES is minimizing the makespan directly, whereas the supervised linear models are learning to discriminate optimal versus suboptimal features sets that implies a better deviation from optimality later on.

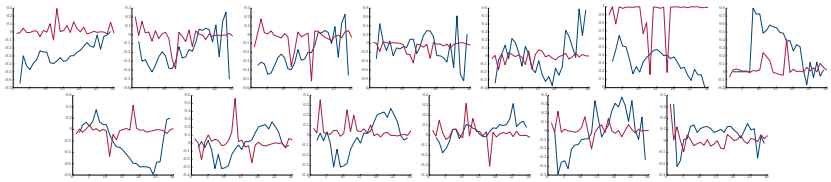

Fig. 8. Linear weights for $\mathcal{P}_1$. Weights (w_1 to w_{13} from left to right, top to bottom) found via CMA-ES optimisation (red), and weights found via learning classification model based on S_b^{cma} (blue).

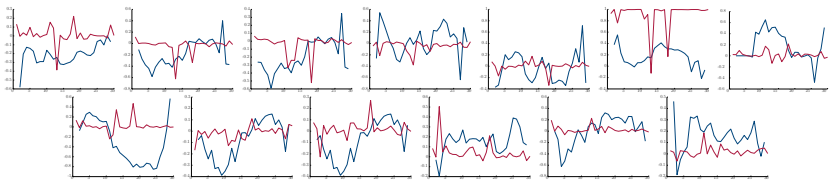

Fig. 9. Linear weights for $\mathcal{P}_2$. Weights (w_1 to w_{13} from left to right, top to bottom) found via CMA-ES optimisation in red, and weights found via learning classification model based on S_b^{cma} in blue.

4.4 Summary and conclusion

As the experimental results illustrate in section 4, the ranking of optimal[3] and suboptimal features are of paramount importance. The subsequent rankings are not of much value, since they are disregarded anyway. However, the trajectories to create training instances have to be varied.

Unlike [8,6,7], learning only on optimal training data was not fruitful. However, inspired by the original work by [5], having heuristic guide the generation of training data, but with nevertheless optimal labelling based on a solver, gave meaningful preference pairs which the learning algorithm could learn. In conclusion, henceforth, the training data will be generate with S_b^{all} scheme for the authors' future work.

Based on these preliminary experiments, we continue to test on a greater variety of problem data distributions for scheduling, namely job-shop and permutation flow-shop problems. Once training data has been carefully created, global dispatching rules can finally be learned, with the hope of implementing them for a greater number of jobs and machines. This is the focus of our current work.

References

1. Panwalkar, S.S., Iskander, W.: A survey of scheduling rules. Operations Research **25**(1) (1977) 45–61
2. Jayamohan, M., Rajendran, C.: Development and analysis of cost-based dispatching rules for job shop scheduling. European Journal of Operational Research **157**(2) (2004) 307–321
3. Ingimundardottir, H., Runarsson, T.P.: Supervised learning linear priority dispatch rules for job-shop scheduling. In Coello, C., ed.: Learning and Intelligent Optimization. Volume 6683 of Lecture Notes in Computer Science. Springer Berlin, Heidelberg (2011) 263–277
4. Hansen, N., Ostermeier, A.: Completely derandomized self-adaptation in evolution strategies. Evol. Comput. **9**(2) (June 2001) 159–195
5. Li, X., Olafsson, S.: Discovering dispatching rules using data mining. Journal of Scheduling **8** (2005) 515–527
6. Malik, A.M., Russell, T., Chase, M., Beek, P.: Learning heuristics for basic block instruction scheduling. Journal of Heuristics **14**(6) (December 2008) 549–569
7. Russell, T., Malik, A.M., Chase, M., van Beek, P.: Learning heuristics for the superblock instruction scheduling problem. IEEE Trans. on Knowl. and Data Eng. **21**(10) (October 2009) 1489–1502
8. Olafsson, S., Li, X.: Learning effective new single machine dispatching rules from optimal scheduling data. International Journal of Production Economics **128**(1) (2010) 118–126

[3] Here the tasks labelled 'optimal' do not necessarily yield the optimum makespan (except in the case of following optimal trajectories), instead these are the optimal dispatches for the given partial schedule.

[3] A note to the reviewer of this paper: we know that some of the figures have too small fonts and are in colour, this will be fixed in the final version (we still have room for 2.5 more pages).

9. Gurobi Optimization, Inc.: Gurobi optimization (version 5.0) [software] (May 2012)
10. Ingimundardottir, H., Runarsson, T.P.: Determining the Characteristic of Difficult Job Shop Scheduling Instances for a Heuristic Solution Method. In Schoenauer, M., ed.: Learning and Intelligent Optimization, 6th International Conference, LION 6, Paris, Springer Lecture Notes in Computer Science (2012)
11. Rosen, K.H.: 9. In: Discrete Mathematics and Its Applications. 5 edn. McGraw-Hill, Inc., New York, NY, USA (2003) 631–700